Energy-Aware Scheduling for Microservice-Based Applications

1 Introduction

Despite the improvements in energy-efficiency of hardware, the overall energy consumption in large-scale computing data centers continues to grow. That increasing energy consumption of computing systems has become a limiting factor for further performance growth due to overwhelming electricity bills, complex requirements for the cooling system and power supply infrastructure, high carbon dioxide footprints, and more [1].

Existing work on energy-aware resource management approaches for data centers mostly relies on reallocating virtual machines between physical hosts, thus consolidating the workload into the minimum of physical resources and preventing resource underutilization [2, 3]. However, little attention is given to application running on these machines and to application-level policies that can facilitate further energy savings.

One substantial portion of cloud-native workloads nowadays is applications built following microservice architectural principles [4, 5, 6, 7]. A microservice-based application is a large distributed system that consists of small, loosely coupled, mono-functional services communicating with each other over a network. The benefits of decomposing a monolithic application into smaller independently deployable and decentralized services include an independent development and deployment of each service, heterogeneity of used tech stacks, loose coupling, and high cohesion. However, as interrelated parts of the application extensively communicate with each other, deployment of services in nodes can greatly affect the energy consumption of the system. In fact, according to data provided by Intel Labs [8], networking accounts for a large fraction of the power consumed by a machine; as such, deploying services with a high communication volume on separate machines will negatively impact the energy consumption of a cluster.

With the growing popularity of microservices, the goal of this project is to propose an energy-aware scheduler for microservice-based applications. The scheduler takes into account information about (a) the traffic patterns between microservices, (b) constraints on resource consumption of each individual microservice, and (c) availability and utilization of nodes in the cluster. It uses this information to adjust the placement of microservices and optimally distribute interrelated microservice workloads. By the end of the project, we intend to measure whether the proposed algorithm improves the utilization of cluster resources and energy-efficiency of the cluster, without sacrificing performance.

We perform our experiments in a cluster managed by Kubernetes [9]. In such clusters, microservices run inside *pods* (typically, one microservice per pod)

and pods are allocated to nodes by the Kubernetes scheduler. There are three major sources of network traffic: traffic generated by application users, traffic generated by communication between pods, and traffic generated by active monitoring (metrics collection). The combination of these three factors can cause congestions in the cluster. When network bandwidth becomes a bottleneck, our proposed algorithm is expected to lower resource utilization and power consumption without sacrificing performance (response time) of applications.

2 Proposed Algorithm

We assume a homogeneous compute cluster consisting of virtual machines (VMs) $\mathbb{VM}$, $|\mathbb{VM}| = v$ which run on physical machines (PMs) $\mathbb{PM}$, $|\mathbb{PM}| = p$, $p \leq v$. Homogeneous cluster implies that all VMs have the same capacity in terms of CPU, memory, and disk, denoted by VM^C, VM^M, and VM^D, respectively. PMs are placed in a number of network racks $\mathbb{R}$, $|\mathbb{R}| = r$, $r \leq p$.

Our goal is to schedule a set of services $\mathbb{S}$ on this compute cluster. For each service $S \in \mathbb{S}$, S^C, S^M, and S^D denote the peak CPU, memory, and disk consumption of S, respectively. These values are initially specified by the user directly, as in the standard Kubernetes scenario. They can further be refined dynamically by observing services' execution over a period of time P.

We assume the standard Kubernetes scheduling strategy of placing new services as they arrive to the cluster based on VMs' capacity. The main issues with this strategy is that (a) it does not re-evaluate and optimize the status of the cluster as new services arrive and the workload of existing services changes and (b) it does not take into account network interactions between the scheduled services.

We thus propose to monitor the current status of the cluster and network interactions between services and, periodically, re-distribute the services in the cluster to achieve better utilization and energy savings. Such re-distribution is required as the workload and interaction patterns between services might change over time, and the initial placement might no longer be optimal.

The goal of our algorithm is to produce a placement that:

- does not exceed the CPU, memory, and disk capacities of each VM (thus maintaining original performance guarantees),
- minimizes network interactions between services placed on different VMs, different PMs, and different racks (thus reducing physical network interactions while improving performance and energy efficiency), and
- "packs" the services on existing VMs (thus ensuring that the overall resource consumption remains low).

An exact solution to this optimization problem is at least NP-complete: the complexity of the well-known *bin packing* problem, which can be used just to place services on VMs, is NP-complete and our problem is at least as hard. As such, in practice, it is unfeasible to efficiently find an optimal solution and we need to apply a heuristic algorithm.

The main idea behind our algorithm is to use Agglomerative Hierarchical Clustering [10, 11] to build "communities" of services that exchange a large volume of information. The clustering algorithm accepts a custom distance

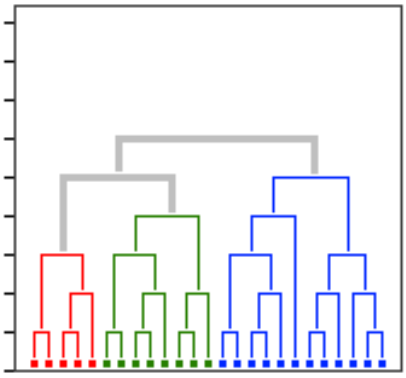

Figure 1: A dendrogram.

function – a measure of similarity between the clustered elements, and builds a dendrogram – a tree diagram arranging the input elements, as show in Figure 1. The root node of the dendrogram represents the entire data set and each leaf is regarded as a data object — in our case, a service. In Figure 1, there are 25 services: 5 in red, 8 in green, and 12 in blue.

The intermediate nodes of a dendrogram describe the proximity of the objects in their subtree to each other: the "lower" the two nodes in the tree are connected, the more similar their corresponding data objects are. In our case, object are considered similar if they exchange a large volume of data with each other. For the example in Figure 1, services depicted in each distinct color are similar to each other, i.e., they exchange a large volume of information with each other. Moreover, services depicted in red are more similar to services depicted in green than those in blue, which means that "red" services exchange more information with "green" services than with "blue" services.

More formally, we denote the cumulative volume of data exchanged between services S_i and S_j over a period of time P by $B_{S_i-S_j}$. For calculating $B_{S_i-S_j}$, we consider incoming and outgoing traffic together, which means $B_{S_i-S_j} = B_{S_j-S_i}$.

We define the similarity (distance) function d between two services S_i and S_j as $d(S_i, S_j) = B_{S_i-S_j}$

For the hierarchical clustering algorithm to work correctly, we also need to provide the distance function between two clusters C_i and C_j, each consisting of multiple services. We define this function as the sum of traffic between all services in C_i with all services in C_j:

$$d(C_i, C_j) = \sum_{\forall S_i \in C_i, \forall S_j \in C_j} B_{S_i-S_j}$$

The goal of our algorithm is to identify clusters of services that can be placed on the same VM, then on other VMs on the same PM, and then on other PMs on the same rack. It accepts as input a dendrogram D, which is a set of nodes $\{N\}$ with the leafs nodes being individual service in $\mathbb{S}$. It then labels each leaf service $S \in \mathbb{S}$ with a $VM \in \mathbb{VM}$ on which this service is scheduled to run.

Algorithm 1 outlines our approach. It maintains a variable cVM which represents the current VM on which the placement is made. Initially, cVM is

set to a random VM from $\mathbb{VM}$ (line 1). The algorithm then starts a pre-order tree traversal of the dendrogram (line 2). That traversal first checks if a node N, which can be a singular service or a group of services, can fit into any VM (line 4). If N is not "placeable", this means that we need to break the group into smaller subsets. The algorithm then recursively visits both children of N by calling traverse($N.left$) and traverse($N.right$). Otherwise, N can be placed in a machine and the goal of the algorithm is to find the "right" machine for all services in N (lines 7–20).

Input : Dendrogram D
Output: A placement of each leaf service $S \in D$ on a $VM \in \mathbb{VM}$

```
1  cVM ← pick random from 𝕍𝕄 ;                    // pick the first VM randomly
2  traverse(D, cVM) ;                            // start the recursive traversal
3  Procedure traverse(N, cVM)
4      if !placeable (N) then
           // N is too big to fit any VM
5          traverse (N.left, cVM);
6          traverse (N.right, cVM);
7      else if placeable (N,cVM) then
           // N can fit the current VM
8          place (N,cVM);
9      else if placeable (N,cVm.PM) then
           // N can fit in a VM from the current PM
10         cVM ← getClosestFromPM(N,cVm.PM) ;   // find the closest VM
            in PM
11         place (N,cVM);
12     else if placeable (N,cVm.PM.Rack) then
           // N can fit in a VM from the current rack
13         cVM ← getClosestFromRack(N,cVm.PM.Rack) ;      // find the
            closest VM in Rack
14         place (N,cVM);
15     else
16         cVM ← getClosestFromSystem(N) ;  // find the closest VM in the
            system
17         if cVM = null then
               // No existing VM can host N
18             return ERROR ;                          // No VM is available!
19         else
20             place (N,cVM);
21     return
```

Algorithm 1: Scheduling Procedure.

The first attempt is to place N on the current VM, cVM (lines 7–8). If cVM does not have enough capacity to host all services in N, the algorithm attempts to find the closest VM in the same PM (lines 9–11). The details of how the heuristics for finding the closest VM works are given below. If, in this step, the algorithm can successfully acquire such a VM on the same PM, all services in N are placed on that VM. Moreover, it becomes the current VM, cVM, to host the following nodes. The rationale for this decision is to try and place subsequently

traversed nodes, which are closest to the current one (in terms of the distance d), on that same VM.

Otherwise, the algorithm attempts to place N in a VM that is on the closest PM in the same rack (lines 12–14), making that VM the current VM to schedule next services on. Finally, if it cannot be placed on a VM in the same rack, the algorithm gets a new rack and switches placing services there (lines 15–20). The details for heuristics we apply for finding the closest PM in a rack and the closest rack in the system are given below. In no suitable VM is found in the entire system, the algorithm terminates and returns an error (line 18).

The algorithm relies on a set of helper functions:

- $label(N, VM)$: labels all leaf services in the sub-tree N of D with the VM and calculates the remaining capacity of the VM

$$VM^{C-remains} \leftarrow VM^{C-remains} - \sum_{\forall S \in N} S_i^C$$

$$VM^{M-remains} \leftarrow VM^{M-remains} - \sum_{\forall S \in N} S_i^M$$

$$VM^{D-remains} \leftarrow VM^{D-remains} - \sum_{\forall S \in N} S_i^D$$

- $isPlaceable(N)$: checks if all leaf services in the sub-tree N of D can run on any VM, i.e., their overall capacity does not exceed the top capacity configured for VMs in the system

$$\sum_{\forall S \in N} S_i^C \leq VM^C \wedge \sum_{\forall S \in N} S_i^M \leq VM^M \wedge \sum_{\forall S \in N} S_i^D \leq VM^D$$

- $isPlaceable(N, VM)$: checks if all leaf services in the sub-tree N of D can run on a particular VM VM, i.e., their overall capacity does not exceed the remaining capacity of that VM

$$\sum_{\forall S \in N} S_i^C \leq VM^{C-remains} \wedge \sum_{\forall S \in N} S_i^M \leq VM^{M-remains} \wedge \sum_{\forall S \in N} S_i^D \leq VM^{D-remains}$$

- $isPlaceable(N, PM)$: checks if all leaf services in the sub-tree N of D can run on a VM in the given PM PM. This function iterates over all VMs in the given PM and returns true if at least one $isPlaceable(N, VM)$ evaluates to true.

- $isPlaceable(N, Rack)$: checks if all leaf services in the sub-tree N of D can run on a VM in the given Rack. Similar to the previous case, this function iterates over all PMs in the given Rack and returns true if at least one $isPlaceable(N, PM)$ evaluates to true.

- $getClosestFromPM(N, PM)$: finds a suitable VM inside the PM. A straightforward solution could be to find a random VM that has enough capacity to host all services in N. However, as our goal is to maximize network interaction between services on one machine and minimize interaction between services across machines, we attempt to find a VM that is

most likely to host a set of services close to N. To achieve that, each PM maintains two data structures: a stack of VMs that already have some services assigned and a set of empty VMs. Each time the algorithm looks for a new VM that can host N, it traverses the stack top down, thus prioritizing VM closer to the top of the stack, as they are more likely to host the "closest" set of nodes. For each traversed VM, it checks whether the VM can host all services in N (by calling $isPlaceable(N, VM)$). If no suitable VM is found in the entire stack, a new VM is allocated from the set of empty VMs, pushed on top of the stack, and returned by the function. The function returns null if no VM is available.

- $getClosestFromRack(N, Rack)$: finds a suitable VM inside the rack. Similar to the previous function, each rack maintains two data structures: a stack of PMs that already have some VMs assigned and a set of empty PMs. Each time the algorithm looks for a new VM that can host N, it traverses the stack top down, thus prioritizing PM closer to the top of the stack, as they are more likely to host the closest set of nodes. For each PM, it calls getVMFromPM(N, PM) and, if a suitable VM found, returns it. If traversal over the stack of "in use" PMs does not find a suitable VM, a new PM is added on top of the stack and a VM is allocated from this PM (by calling getVMFromPM(N, PM) again). The function returns null if no PM is available.

- $getClosestFromSystem(N)$: finds any suitable VM in the cluster. This function is called if N could not be placed in the same VM, PM, and rack as the previously placed node. Then, it finds and returns a new rack containing a VM that can host all services in N (using the same stack-based logic as in the previous function) or returns null if none found.

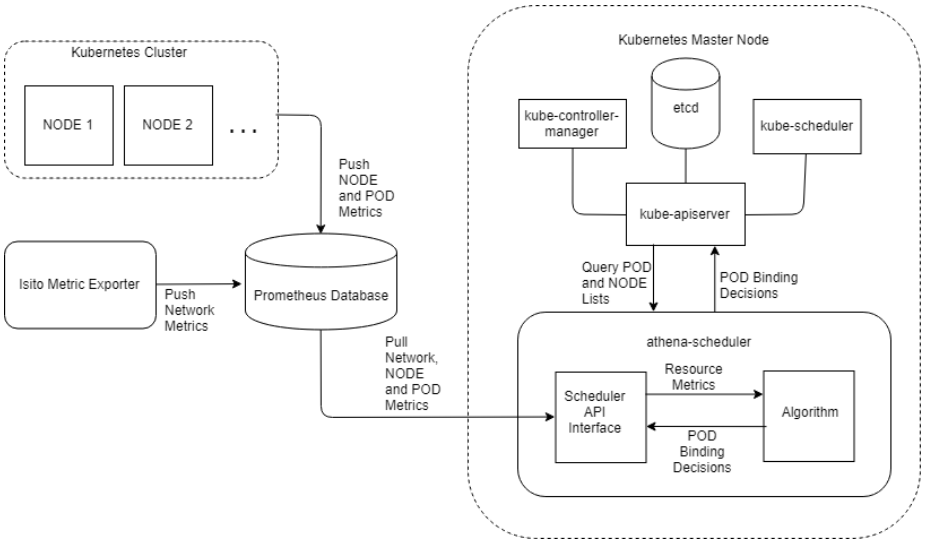

Figure 2: Prototype Design

3 Prototype Implementation

Our custom scheduler is configured to run on Kubernetes and uses Prometheus, Grafana and Istio, which are described below.

3.1 Tool Selection

Kubernetes. Kubernetes is an open-source orchestration tool for managing containerized applications across multiple hosts [9]. It consists of at least one master node and several worker nodes. Together, the master and worker nodes form a Kubernetes cluster. The Kubernetes master runs three processes to maintain the desired state of the cluster: kube-apiserver, kube-controller-manager, and kube-scheduler (see Figure 2).

- The kube-apiserver acts as the front end to the cluster's shared state and exposes REST APIs through which different components can interact.
- The kube-controller-manager is a control loop that watches the state of the cluster through kube-apiserver. It works towards moving the current state of the cluster to the desired state by performing a variety of tasks such as starting or stopping containers, scaling the number of replicas of a given application, and more.
- The kube-scheduler (a.k.a. the default scheduler) is a process that schedules pods onto worker nodes using the information about cluster topology and application workload.

Each worker node runs two processes: kubelet, which communicates with the master, and kube-proxy – a network proxy. Kubernetes also has a database called etcd that stores and replicates the cluster state. Kubernetes makes use of abstractions to create the components of the distributed application and their associated resources. These abstractions are represented by objects in the Kubernetes API. Some of the Kubernetes objects are Pods, Services, Volumes, Namespaces, Deployments, and DaemonSets. The objects have a resourceVersion that represents the version of the object at a given instance as stored in the Kubernetes database (etcd). Kubernetes also offers a command line interface called Kubectl that connects to the kube-apiserver for managing applications on the cluster.

Prometheus and Grafana. To measure the real-time resource usage of containers, pods, and nodes, we deployed a full metrics collection and monitoring pipeline using Prometheus and Grafana. Prometheus is an open-source monitoring and alerting framework that scrapes core metrics (CPU, DISK I/O, Network, Memory) from various Kubernetes endpoints via a pull model over HTTP and writes them to a time series database. Prometheus server runs as a pod in the cluster and it offers an HTTP API that is reachable under /api/v1 to retrieve the desired metrics. Grafana is a data visualization and monitoring tool that can be integrated to query Prometheus and to create dashboards for exploring and understanding the metrics Prometheus collects.

Istio. Istio is an open-source platform that connects and manages microservices by creating a network service mesh. It acts as a network proxy and is deployed as a sidecar to each microservice in the Kubernetes pod without any changes

to the source code. It offers functionality like network monitoring, load balancing, fault injection, security and tracing. Istio gathers vital metrics like the number of bytes exchanged between microservices, request duration and other telemetry data for every hop. It comes with a built in Prometheus adapter that exposes these metrics and the Prometheus server scrapes these metrics from these exposed endpoints.

3.2 Scheduler Implementation

Kubernetes supports configuring multiple schedulers, making it possible to choose the appropriate scheduler at the time of deployment. Our custom scheduler runs on the master node outside the cluster as a process (see Figure 2) and has an API interface to connect and interact with both the kube-apiserver and Prometheus server. The scheduling algorithm takes into consideration three factors to schedule a pod to a node, which are: (a) the constraints on resource consumption of the pod that can be read from Pod specifications given at the time of deployment, (b) cluster resource availability and utilization details that are obtained from Prometheus or Kubernetes metrics API, and (c) Network traffic patterns between deployed pods obtained from Prometheus that are collected and pushed by Istio. Once a scheduling decision is made, the decision is propagated back to the kube-apiserver via a binding API that binds the pod to a node.

Kuberentes APIs. The API interface uses a Kubernetes API client library in Golang called 'client-go' [12], to implement API calls to the kube-apiserver. These APIs give the list of objects, such as Pods and Nodes, available in the cluster. The meta-data and specifications of the objects provide user-defined parameters like requested resources. The parameters of interest here are CPU, memory, and disk limits of Pods. Several API endpoints are used to perform tasks like listing available nodes, listing pods to be scheduled, and propagating scheduling decisions. Table 1 provides a list of API calls we used and describes the functionality they provide.

Table 1: Kubernetes APIs and their functionalities

Functionality and Metric	**API Endpoint and Resource Type**
List available nodes ($\mathbb{VM}$)	'/api/v1/{namespaces}/nodes'
List pods to be scheduled ($\mathbb{S}$)	'/api/v1/{namespaces}/pods'
Propagate scheduling decision	'/api/v1/namespaces/{namespace}/bindings'
Retrieve pod CPU limit (S^C)	Pod.Spec.Containers.Resouces.Limits[ResouceCPU]
Retrieve pod memory limit (S^M)	Pod.Spec.Containers.Resouces.Limits[ResouceMemory]
Retrieve pod disk limit (S^D)	Pod.Spec.Containers.Resouces.Limits[ResouceStorage]

Prometheus APIs. The API interface also uses a Prometheus HTTP library called 'client-golang' [13] to query metrics from Prometheus database. The metrics retrieved from Prometheus and their corresponding queries are listed in Table 2.

Communication Pattern Matrix. Our scheduler takes in the network traffic pattern between the microservices as input to perform agglomerative hierarchi-

Table 2: Metrics from Prometheus Query

Metric	Query Parameter
Node CPU capacity (VM^C)	machine_cpu_cores
Node memory Capacity (VM^M)	machine_memory_bytes
Node lables (PM and $Rack$)	machine_memory_bytes

cal clustering and make the binding decision. The traffic pattern is represented by a half matrix with the height and width of the matrix equal to the number of microservices deployed in a particular namespace. Each microservice is assigned an index and represents the row and column of the matrix corresponding to their index. Any value in the matrix represents the total volume of bytes exchanged between the corresponding microservices. The value is obtained by querying two Istio metrics from Prometheus (incoming and outgoing traffic) and computing their sum. The metrics have source workload and destination workload labels that are used to query the volume of bytes exchanged between any two particular microservices. They metrics used are listed in Table 3.

Table 3: Metrics from Istio

Metric	Query Parameter
Incoming traffic of services ($B_{S_i-S_j}$)	istio_tcp_received_bytes_total
Outgoing traffic of services ($B_{S_i-S_j}$)	istio_tcp_sent_bytes_total

Scheduler. The scheduler itself is implemented in Golang and Python using 1079 lines of code. It used the implementation of Agglomerative Hierarchical Clustering from the Python scipy.cluster.hierarchy.linkage library [14].

4 Evaluation Methodology

We evaluate the performance (in terms of response time) of the scheduled applications and the overall energy consumption of the cluster for our algorithm described in Section 2 and compare with the default, out of the box scheduling algorithm provided by Kubernetes. We run the experiments on a benchmark application and on an openly-available microservice-based application: Robot Shop [15]. We describe the details of our experimental setup below.

4.1 Cluster Configuration

For our experiments, we use four physical machines (servers); each server has 4 cores, 8 GB of memory, 1 TB disk space, 1 Gbps NIC, and runs Ubuntu 16.04. To simulate a cluster with VMs, PMs, and racks using these five machines, we treat each of them as a VM server and then "group" them into PMs and racks by controlling the network bandwidth between the machines. Specifically, we

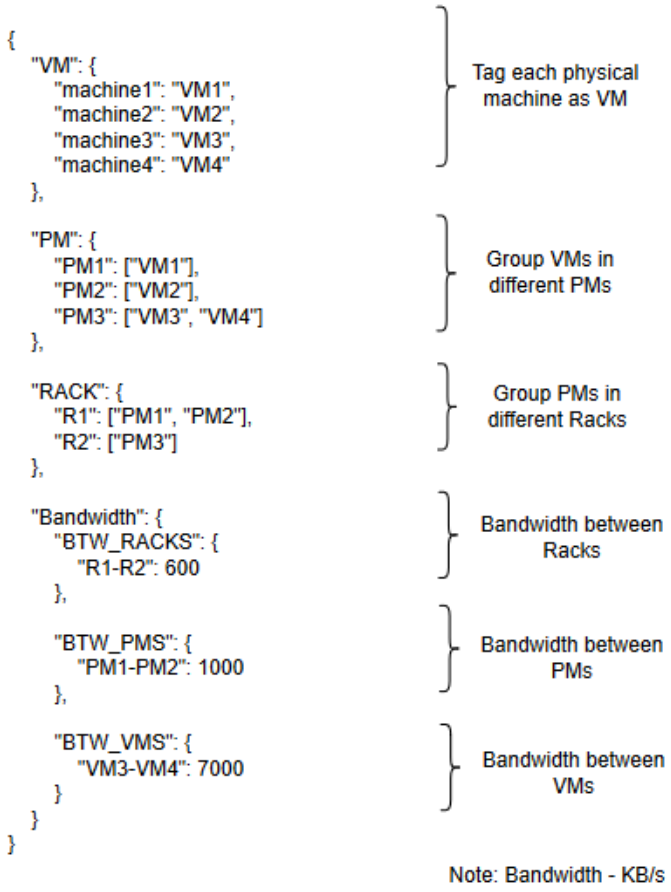

Figure 3: Cluster simulation sample input

tag each machine as a VM, naming them VM1, VM2, VM3, and VM4. Then, we divide them into three groups and tag group as PM1, PM2 and PM3. Further, we divide these PMs into two different racks, R1 and R2.

To simulate networking bandwidth in the cluster, we configure the network bandwidth between VMs, PMs and Racks using Traffic Control in the Linux kernel. With this type of flexible configuration mechanism, it is easy to simulate different cluster set ups to perform experiment. Figure 3 shows the JSON input file used to define the cluster. In our case, VM1 is placed on PM1, VM2 is placed on PM2, and both VM3 and VM4 are placed on PM3. Both PM1 and PM2 are in R1, while PM3 is in R2. The network bandwidth restriction are also given at the bottom of this configuration file, in KBps. For example, the network bandwidth between VM3 and VM4 is 7000 KBps, while the bandwidth between VM1 and VM2, which are in two different PMs, PM1 and PM2, is 1000 KBps.

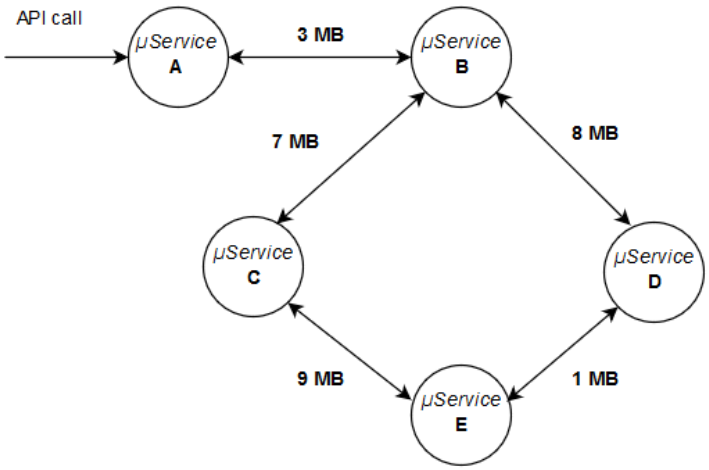

Figure 4: Communication pattern in benchmark application

4.2 Case Studies

4.2.1 Benchmark application

We designed a benchmark application that consists of five microservices, as shown in Figure 4. The figure also depicts the amount of data exchanged between the microservices, in megabytes. We intentionally vary the amount of data exchanged between the microservices, to make sure that our scheduler considers the communication patterns when making placement decision. We use this application to stress-test our algorithm.

4.2.2 Robot Shop

Robot Shop [15] is an openly-available sample e-commerce application that consists of 12 microservices: web, user, mongodb, catalogue, cart, redis, dipatch, rabbitmq, ratings, shipping, mysql, and payment. The data exchanges varies between the microservices, as seen in Figure 5. Thus, this application is highly suitable for evaluating our scheduler. We use this application to show the applicably of our algorithm in practice.

For the evaluation, we have chosen to trigger the API of the cart microservice, "/api/cart/{user}/{product}/{quatity}", that interacts with Redis, catalouge, and mongodb to complete the request.

4.3 Load generation

We use sinusoidal pattern to generate load on microservices. Previous studies of user load for the real-world applications, such as Wikipedia, showed that user load follows that pattern [16]. We simulate the varying load on the benchmarking application and Robot Shop by increasing the number of concurrent requests with time. As bandwidth usage and network bottlenecks increase with number of concurrent requests, this method made it possible for us to evaluate the performance of our custom scheduler under varying network usage.

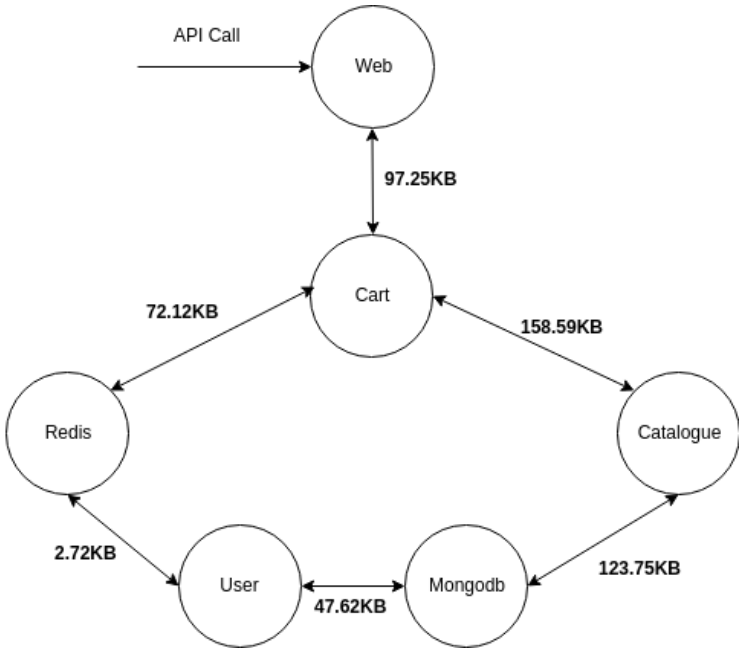

Figure 5: Communication pattern in Robot Shop application

We used Apache HTTP server benchmarking tool [17] for simulating sinusoidal load pattern on benchmarking application, increasing the load incrementally, from 1 to 10 concurrent requests, and then decreasing it in the similar manner, as shown in Figure 6.

For the Robot Shop application, we generated the load using Jmeter [18] by creating a test plan in the Jmeter that consists of 18 thread groups and each thread group contains different number of parallel HTTP requests. These thread groups are executed sequentially, starting with 10 concurrent requests and increasing up to 50 concurrent requests in interval of 10 requests, as shown in the Figure 9.

4.4 Metrics and Measures

We perform the experiments on the two applications described above and record the response time for requests sent to the application and energy consumption of the machines in the cluster. We follow below steps for each application:

1. Deploy an application with Kubernetes default scheduler.
2. Put varying loads on the deployed application using a load generator.
3. For each load pattern, record the response time from the load generator and energy consumption using external power meter.
4. Repeat the steps 1,2, and 3 for Kubernetes configured with our custom scheduler.

5 Results

In the section, we discuss the results of our experiments.

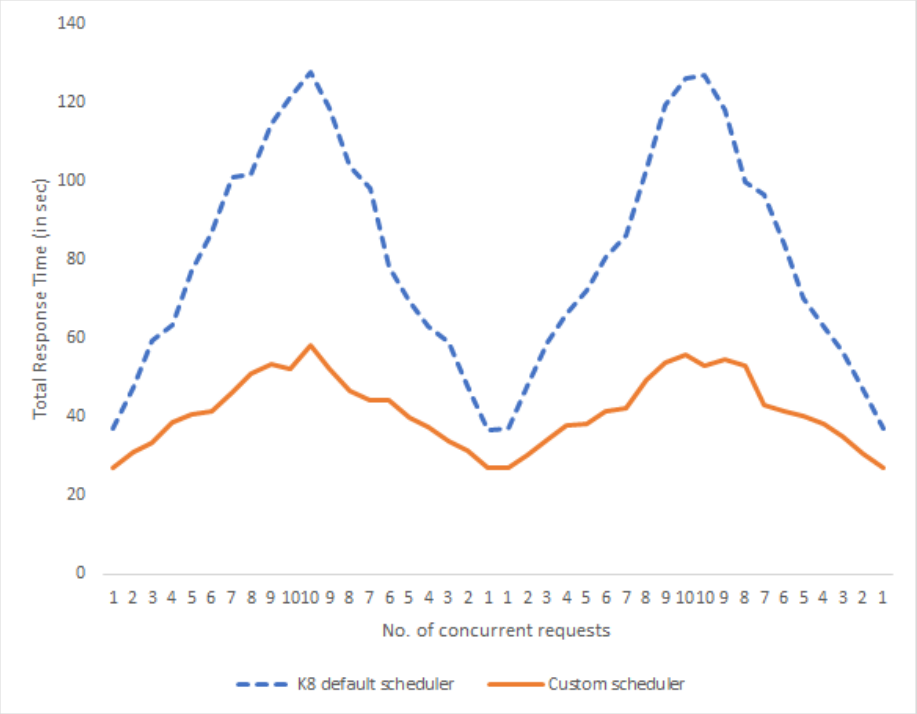

Figure 6: Performance comparison of schedulers for Benchmark application

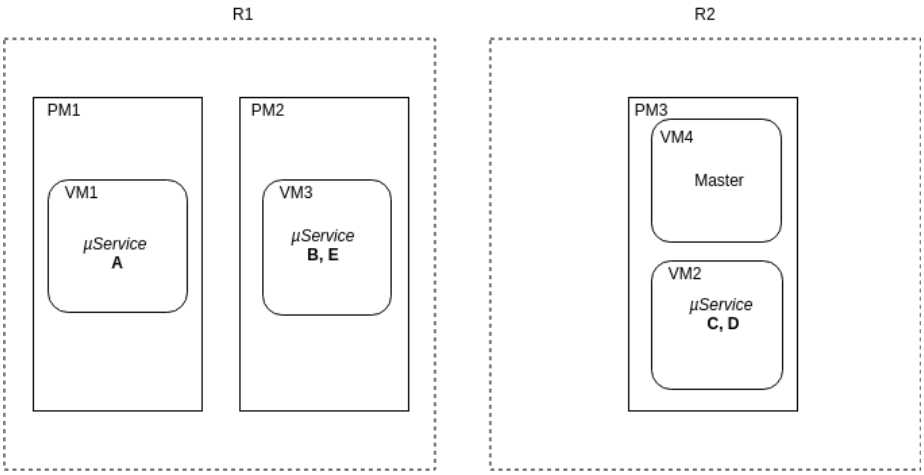

Figure 7: Bemchmark application microservices placement by Kubernetes default scheduler

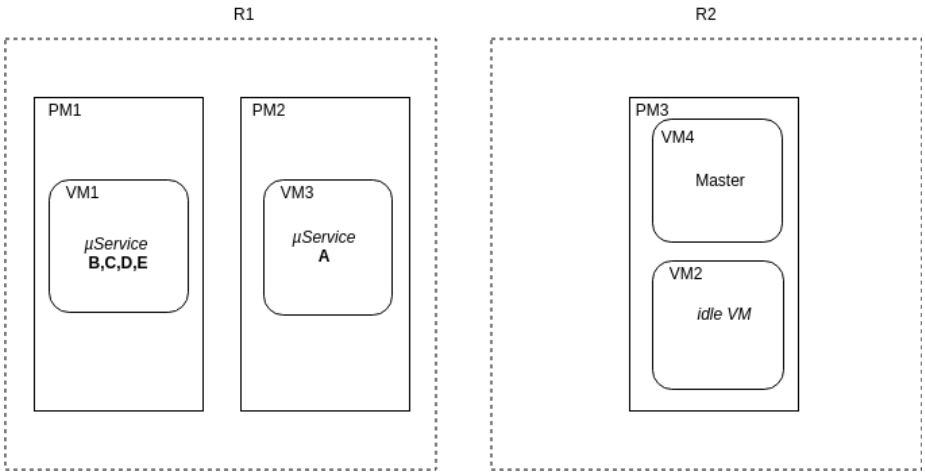

Figure 8: Benchmark application microservices placement by our custom scheduler scheduler

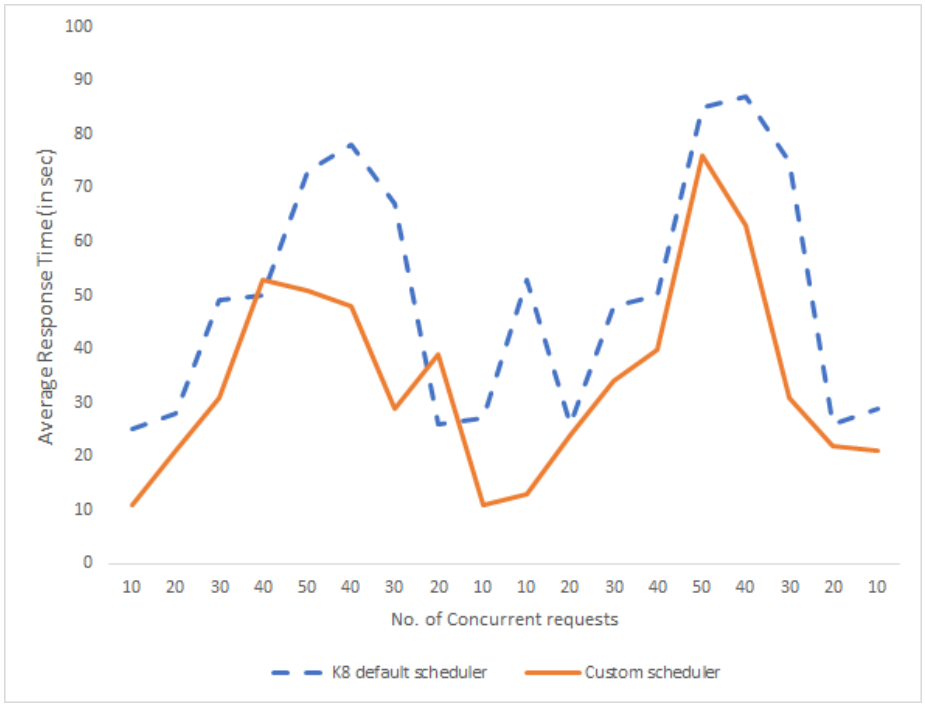

Figure 9: Performance comparison of schedulers for Robot Shop application

5.1 Evaluation on our benchmark application

First, we deploy the benchmark application using the Kubernetes default scheduler and apply the load on the deployed application. We perform the same experiment using our custom scheduler. The performance obtained for both schedulers is shown in the Figure 6.

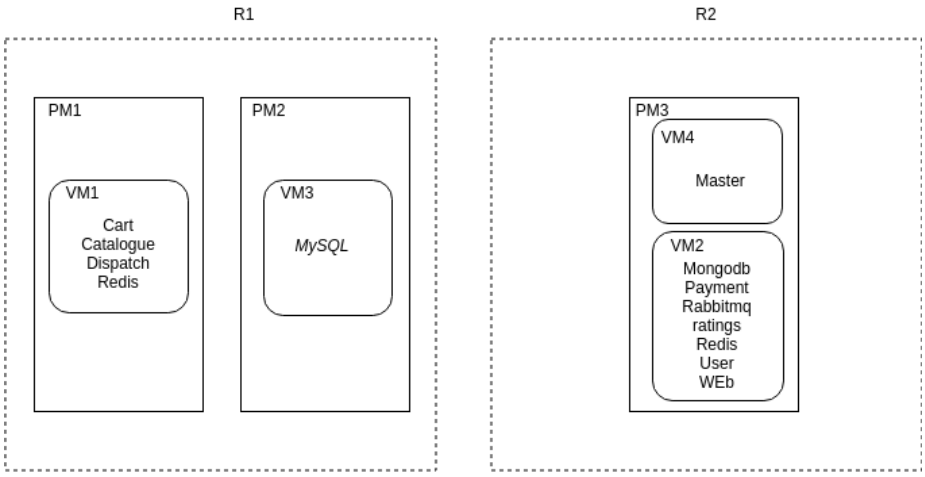

Figure 10: Robot Shop application microservices placement by Kubernetes default scheduler

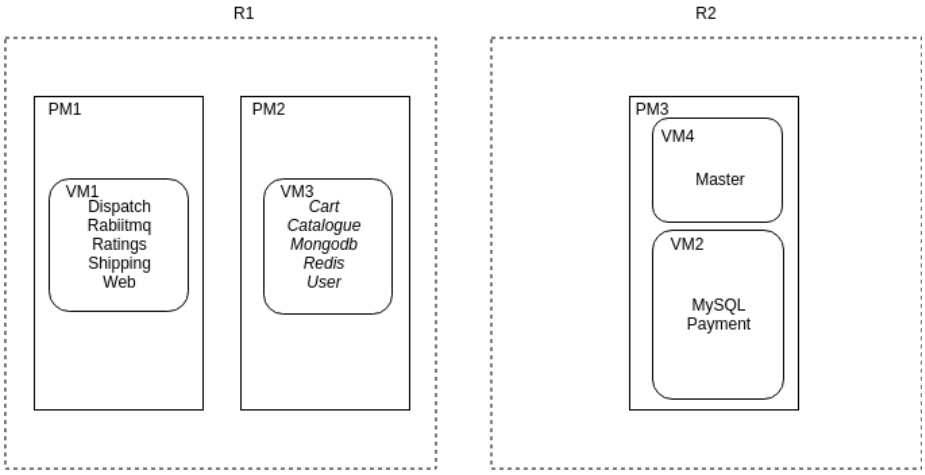

Figure 11: RobotShop application microservices placement by our custom scheduler scheduler

We deployed using our custom scheduler, the application consistently outperforms its equivalent deployed using the Kubernetes default scheduler. On average, performance of the benchmark application improves by approximately 48% and we get energy saving of around 42% at the completion of the experiment. We get the improvement in the performance and energy consumption in case of our scheduler because the scheduler also takes into consideration the communication patterns among the microservices and places the highly communicating microservices as close as possible. Figures 7 and 8 show the placement of the microservices in the cluster by the default scheduler and our custom scheduler, respectively.

5.2 Evaluation on Robot Shop application

Similar to the benchmark app, we first deploy Robot Shop using the Kubernetes default scheduler and apply the load on the deployed application as described in the load generation. We then perform the same experiment using our custom scheduler. The performance for both schedulers is shown in the Figure 9.

Again, the performance of Robot Shop application deployed using our custom scheduler is better than its equivalent deployed using the Kubernetes default scheduler. On average, performance of the application improves by approximately 30% and the energy saving is around 34% for the completion of the experiment. Figures 10 and 11 show the placement of the microservices in the cluster by the default scheduler and our custom scheduler, respectively.

References

[1] A. Beloglazov, R. Buyya, Y. C. Lee, and A. Y. Zomaya, "A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems," *Advances in Computers*, vol. 82, pp. 47–111, 2011.

[2] A. Beloglazov, J. H. Abawajy, and R. Buyya, "Energy-aware Resource Allocation Heuristics for Efficient Management of Data Centers for Cloud Computing," *Future Generation Comp. Syst.*, vol. 28, no. 5, pp. 755–768, 2012.

[3] A. J. Younge, G. von Laszewski, L. Wang, S. Lopez-Alarcon, and W. Carithers, "Efficient Resource Management for Cloud Computing Environments," in *Proc. of International Green Computing Conference 2010*, 2010, pp. 357–364.

[4] C. Richardson, "Pattern: Microservice Architecture." [Online]. Available: http://microservices.io/patterns/microservices.html

[5] J. Lewis and M. Fowler, "Microservices: A Definition of This New Architectural Term." [Online]. Available: https://martinfowler.com/articles/microservices.html

[6] T. Mauro, "Adopting Microservices at Netflix: Lessons for Architectural Design." [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[7] J. Sloyer, "Microservices in Bluemix." [Online]. Available: https://www.ibm.com/blogs/bluemix/2015/01/microservices-bluemix/

[8] L. Minas and B. Ellison, *Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers.* Intel Press, 2009.

[9] Kubernetes, "Kubernetes," last accessed: October 2018. [Online]. Available: http://kubernetes.io/

[10] S. C. Johnson, "Hierarchical Clustering Schemes," *Psychometrika*, vol. 32, no. 3, pp. 241–254, 1967.

[11] R. Xu and D. Wunsch, II, "Survey of Clustering Algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005.

[12] Kubernetes, "Go client for kubernetes." last accessed: November 2018. [Online]. Available: https://github.com/kubernetes/client-go

[13] Prometheus, "Prometheus instrumentation library for go applications," last accessed: November 2018. [Online]. Available: https://github.com/prometheus/client_golang

[14] SciPy, "scipy.cluster.hierarchy.linkage," last accessed: November 2018. [Online]. Available: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.cluster.hierarchy.linkage.html

[15] Instana, "Robot-shop," last accessed: October 2018. [Online]. Available: https://github.com/instana/robot-shop

[16] G. Urdaneta, G. Pierre, and M. Van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.

[17] "ab - apache http server benchmarking tool - apache http server version 2.4," https://httpd.apache.org/docs/2.4/programs/ab.html, (Accessed on 12/02/2018).

[18] "Apache jmeter," https://jmeter.apache.org/, (Accessed on 12/02/2018).